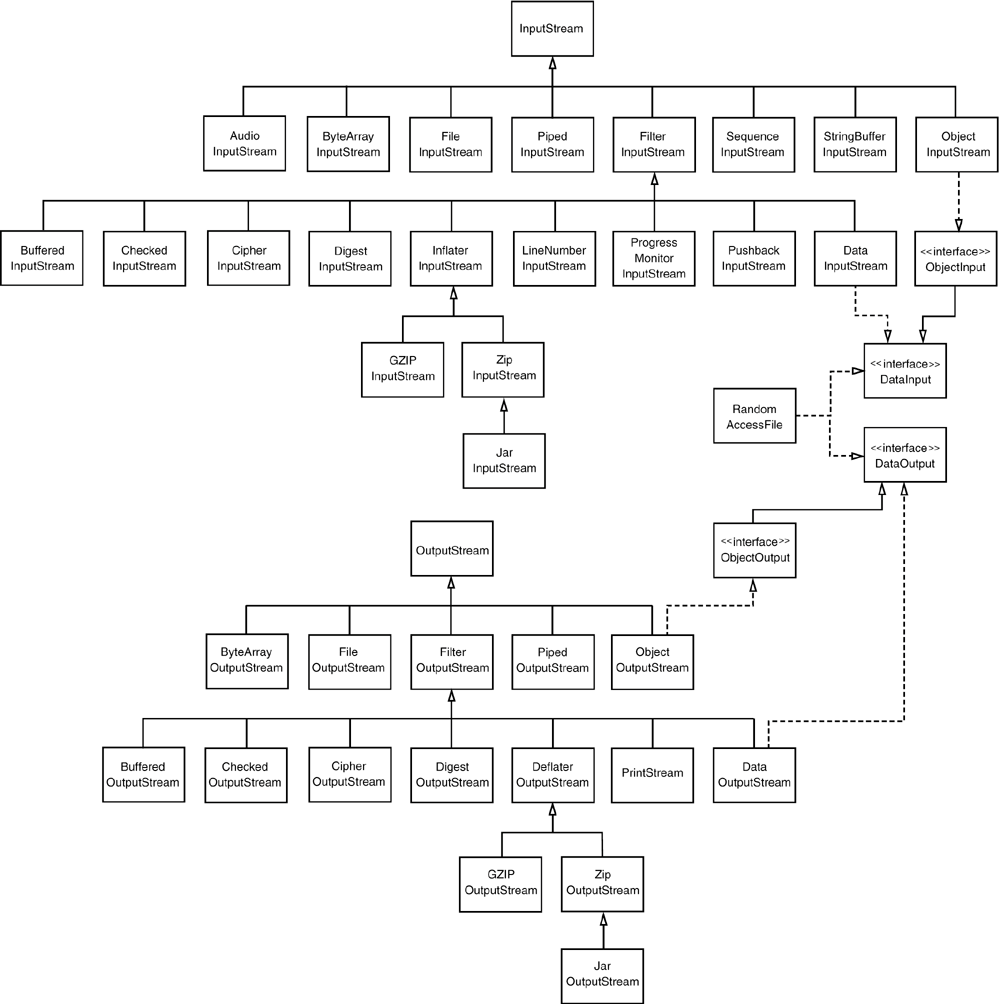
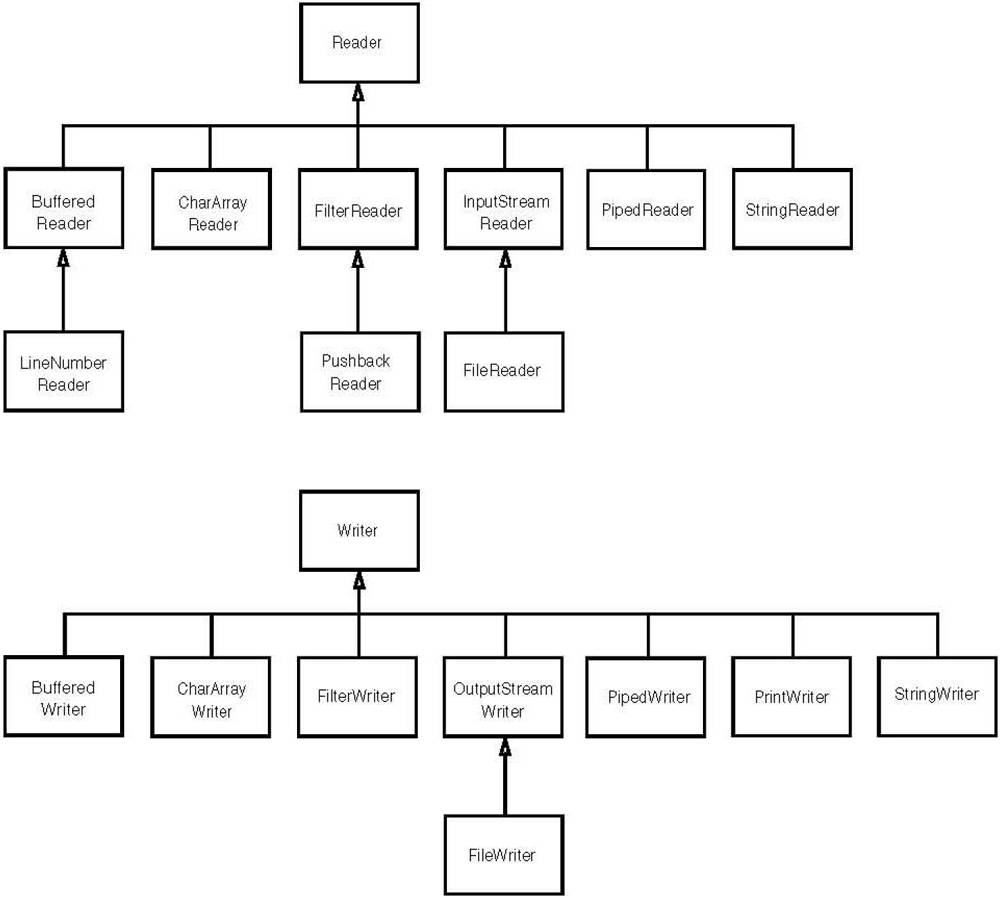
**Java IO**

       本篇主要讲述IO相关的内容，主要包括：与IO相关的简单的历史背景知识；Java IO的简单分类；与IO设计相关的两个模式；同时列举几个简单的例子；分析其中几个实现的源代码；最后给大家一些简单扩展的例子。治学先治史，下面我们先从简单的历史开始吧！





## 一、历史背景

       “对语言设计人员来说，创建好的输入／输出系统是一项特别困难的任务。” ――《Think in Java》

       无论是系统、还是语言的设计中IO的设计都是异常复杂的。面临的最大的挑战一般是如何覆盖所有可能的因素，我们不仅仅要考虑文件、控制台、网络、内存等不同的种类，而且要处理大量的不同的读取方式，如：顺序读取、随机读取，二进制读取、字符读取，按行读取、按字符读取……

       Linux是第一个将设备抽象为文件的操作系统，在Linux中所有的外部设备都可以用读取文件的方法读取，这样编程人员就可以以操作文件的方法操作任何设备。C++在IO方面也做了一些改进――引进了流的概念，我们可以通过cin、cout读写一些对象。Java语言在IO设计方面取得较大的成功，它是完全面向对象的，主要采用装饰器模式避免大量的类，包括了最大的可能性，提供了较好的扩展机制……

       “Java库的设计者通过创建大量类来攻克这个难题。事实上，Java的IO系统采用了如此多的类，以致刚开始会产生不知从何处入手的感觉（具有讽刺意味的是，Java的IO设计初衷实际要求避免过多的类）。” 上面一段来自《Think in Java》，确实很多初学者刚刚学习java的IO时会比较茫然，不过等我们知道装饰器模式（Decorator）的用意、场景及其在Java的IO包中的使用，你可能会真正领会整个IO的FrameWork。

## 二、IO的分类

    Java IO一般包含两个部分：1.java.io包中堵塞型IO；2.java.nio包中的非堵塞型IO，通常称为New IO。学过操作系统的朋友都知道系统运行的瓶颈一般在于IO操作，一般打开某个IO通道需要大量的时间，同时端口中不一定就有足够的数据，这样read方法就一直等待读取此端口的内容，从而浪费大量的系统资源。有人也许会提出使用java的多线程技术啊！但是在当前进程中创建线程也是要花费一定的时间和系统资源的，因此不一定可取。Java New IO的非堵塞技术主要采用了Observer模式，就是有一个具体的观察者和＝监测IO端口，如果有数据进入就会立即通知相应的应用程序。这样我们就避免建立多个线程，同时也避免了read等待的时间。不过本篇主要讲述java的堵塞型IO，就是我们通常应用的那个包。

打开你的java.io包你可以看到Java的IO包含大量的类和接口（JDK1.6中包含83个类或者接口），如此众多的类和接口似乎无从下手。下面就将IO简单地分类。Java的IO主要包含三个部分：

1.流式部分----IO的主体部分；

2.非流式部分----主要包含一些辅助流式部分的类，如：File类、RandomAccessFile类和FileDescriptor等类；

3.文件读取部分的与安全相关的类，如：SerializablePermission类。以及与本地操作系统相关的文件系统的类，如：FileSystem类和Win32FileSystem类和WinNTFileSystem类。

       流式部分可以概括为：两个对应一个桥梁。两个对应指：1.字节流（Byte Stream）和字符流（Char Stream）的对应；2.输入和输出的对应。一个桥梁指：从字节流到字符流的桥梁。对应于输入和输出为InputStreamReader和OutputStreamWriter。

       在流的具体类中又可以具体分为：

1.介质流（Media Stream或者称为原始流Raw Stream）----主要指一些基本的流，他们主要是从具体的介质上，如：文件、内存缓冲区（Byte数组、Char数组、StringBuffer对象）等，读取数据；

2.过滤流（Filter Stream）----主要指所有FilterInputStream/FilterOutputStream和FilterReader/FilterWriter的子类，主要是对其包装的类进行某些特定的处理，如：缓存等。

## 三、IO中的流

       流具有最基本的特点：“One dimension , one direction .” 即流是一维的，同时流是单向的。关于维和我们通常说的一维长度，二维平面，三维空间，四维时空……是同一个概念，流就是一维的。单向就是只可以一个方向（按顺序从头至尾依次）读取，不可以读到某个位置，再返回前面某个位置。流的概念和实际水流的概念基本一致，水只可以从高向低一个方向流动。我们某时在目地喝了一口水，下次在同一个地点喝水已经不是当时的那片水了。

       流的这种特性在JMS（Java Message Service）的API设计中得到了体现。JMS是J2EE平台下面向消息中间件的一个标准。（关于中间件技术有机会和大家探讨）JMS中有五种具体类型的消息，这些消息一般分为两类：1.流式的消息----包含ByteMessage和StreamMessage；2.非流式的消息――包含TextMessage、ObjectMessage和MapMessage。我们在明白IO中流的特点后，基本可以明白JMS API设计者的意图。

       可能有些场合我们需要在文件中随机插入数据、在流中来来回回地执行某些操作，这时候我们绝对不可以使用流相关的对象。很幸运JDK的设计者为我们设计了一个单独的类RandomAccessFile，它可以完成打开、关闭文件、以基本数据类型的方式读取数据、读取下一个行、以UTF等格式读取数据、写入各种类型的数据、比较特殊的是他可以通过文件指针的seek方法让文件指针移到某个位置，可以通过[getFilePointer方法得到当前指针的位置、可以通过length（）方法得到当前文件的容量、通过](mk:@MSITStore:E:%20Java%20%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%B9%A6%E7%B1%8D%20%E4%B9%B1%E4%B8%83%E5%85%AB%E7%B3%9F%20jdk150.chm::/jdk150/api/java/io/RandomAccessFile.html#getFilePointer())[getFD得到](mk:@MSITStore:E:%20Java%20%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%B9%A6%E7%B1%8D%20%E4%B9%B1%E4%B8%83%E5%85%AB%E7%B3%9F%20jdk150.chm::/jdk150/api/java/io/RandomAccessFile.html#getFD())[FileDescriptor对象，通过](mk:@MSITStore:E:%20Java%20%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%B9%A6%E7%B1%8D%20%E4%B9%B1%E4%B8%83%E5%85%AB%E7%B3%9F%20jdk150.chm::/jdk150/api/java/io/FileDescriptor.html)[getChannel方法得到](mk:@MSITStore:E:%20Java%20%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%B9%A6%E7%B1%8D%20%E4%B9%B1%E4%B8%83%E5%85%AB%E7%B3%9F%20jdk150.chm::/jdk150/api/java/io/RandomAccessFile.html#getChannel())[FileChannel对象，从而和New IO整合。](mk:@MSITStore:E:%20Java%20%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%B9%A6%E7%B1%8D%20%E4%B9%B1%E4%B8%83%E5%85%AB%E7%B3%9F%20jdk150.chm::/jdk150/api/java/nio/channels/FileChannel.html)

       下面比较简单的分析IO中的各个对象吧！

### 3.1 IO中的输入字节流

       下面是IO中输入字节流的继承图。

o        InputStream

o      ByteArrayInputStream

o        FileInputStream

o        FilterInputStream

o        BufferedInputStream

o        DataInputStream

o        LineNumberInputStream

o        PushbackInputStream

o        ObjectInputStream

o        PipedInputStream

o        SequenceInputStream

o        StringBufferInputStream

在上面的关系图中可以看出：  
1.InputStream是所有的输入字节流的父类，它是一个抽象类。  
2. ByteArrayInputStream、StringBufferInputStream、FileInputStream是三种基本的介质流，它们分别将Byte数组、StringBuffer、和本地文件中读取数据。PipedInputStream是从与其它线程共用的管道中读取数据，与Piped相关的知识会用专门的一小节讲解。  
3. ObjectInputStream和所有FilterInputStream的子类都是装饰流（装饰器模式的主角）。下表列出了这些流的功能及如何使用它们（具体使用在讲解完装饰器模式后会举几个例子）。

基本输入字节流：

|  |  |  |
| --- | --- | --- |
| 类 | 功能 | 如何构造 |
| 怎样使用 |
| ByteArrayInputStream | 将内存中的Byte数组适配为一个InputStream。 | 从内存中的Byte数组创建该对象（2种方法） |
| 一般作为数据源，会使用其它装饰流提供额外的功能，一般都建议加个缓冲功能。 |
| StringBufferInputStream | 将内存中的字符串适配为一个InputStream。 | 从一个String对象创建该对象。底层的实现使用StringBuffer。该类被Deprecated。主要原因是StringBuffer不应该属于字节流，所以推荐使用StringReader。 |
| 一般作为数据源，同样会使用其它装饰器提供额外的功能。 |
| FileInputStream | 最基本的文件输入流。主要用于从文件中读取信息。 | 通过一个代表文件路径的 String、File对象或者 FileDescriptor对象创建。 |
| 一般作为数据源，同样会使用其它装饰器提供额外的功能。 |
| PipedInputStream | 读取从对应PipedOutputStream写入的数据。在流中实现了管道的概念。 | 利用对应的PipedOutputStream创建。 |
| 在多线程程序中作为数据源，同样会使用其它装饰器提供额外的功能。 |
| SequenceInputStream | 将2个或者多个InputStream 对象转变为一个InputStream. | 使用两个InputStream 或者内部对象为InputStream 的Enumeration对象创建该对象。 |
| 一般作为数据源，同样会使用其它装饰器提供额外的功能。 |
| FilterInputStream | 给其它被装饰对象提供额外功能的抽象类 | 主要子类见下表 |
|  |

装饰、输入字节流：

|  |  |  |
| --- | --- | --- |
| 类 | 功能 | 如何构造 |
| 怎样使用 |
| DataInputStream | 一般和DataOutputStream配对使用,完成基本数据类型的读写。 | 利用一个InputStream构造。 |
| 提供了大量的读取基本数据类新的读取方法。 |
| BufferedInputStream | 使用该对象阻止每次读取一个字节都会频繁操作IO。将字节读取一个缓存区，从缓存区读取。 | 利用一个InputStream、或者带上一个自定义的缓存区的大小构造。 |
| 使用InputStream的方法读取，只是背后多一个缓存的功能。设计模式中透明装饰器的应用。 |
| LineNumberInputStream | 跟踪输入流中的行号。可以调用getLineNumber( )和 setLineNumber(int)方法得到和设置行号。 | 利用一个InputStream构造。 |
| 紧紧增加一个行号。可以象使用其它InputStream一样使用。 |
| PushbackInputStream | 可以在读取最后一个byte 后将其放回到缓存中。 | 利用一个InputStream构造。 |
| 一般仅仅会在设计compiler的scanner 时会用到这个类。在我们的java语言的编译器中使用它。很多程序员可能一辈子都不需要。 |

### 3.2 IO中的输出字节流

       下面是IO中输出字节流的继承图。

* + OutputStream
    - ByteArrayOutputStream
    - FileOutputStream
    - FilterOutputStream
      * BufferedOutputStream
      * DataOutputStream
      * PrintStream
    - ObjectOutputStream
    - PipedOutputStream

在上面的关系图中可以看出：1.OutputStream是所有的输出字节流的父类，它是一个抽象类。2. ByteArrayOutputStream、FileOutputStream是两种基本的介质流，它们分别向Byte数组、和本地文件中写入数据。PipedOutputStream是向与其它线程共用的管道中写入数据， 3. ObjectOutputStream和所有FilterOutputStream的子类都是装饰流。下表列出了输出字节流的功能及如何使用它们。

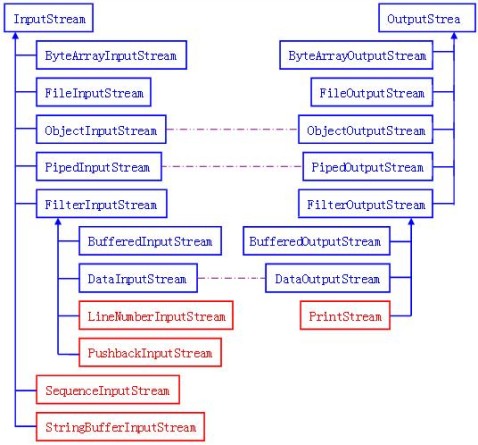
|  |  |  |
| --- | --- | --- |
| 类 | 功能 | 如何构造 |
| 怎样使用 |
| ByteArrayOutputStream | 在内存中创建一个buffer。所有写入此流中的数据都被放入到此buffer中。 | 无参或者使用一个可选的初始化buffer的大小的参数构造。 |
| 一般将其和FilterOutputStream套接得到额外的功能。建议首先和BufferedOutputStream套接实现缓冲功能。通过toByteArray方法可以得到流中的数据。（不通明装饰器的用法） |
| FileOutputStream | 将信息写入文件中。 | 使用代表文件路径的String、File对象或者 FileDescriptor对象创建。还可以加一个代表写入的方式是否为append的标记。 |
| 一般将其和FilterOutputStream套接得到额外的功能。 |
| PipedOutputStream | 任何写入此对象的信息都被放入对应PipedInputStream 对象的缓存中，从而完成线程的通信，实现了“管道”的概念。具体在后面详细讲解。 | 利用PipedInputStream构造 |
| 在多线程程序中数据的目的地的。一般将其和FilterOutputStream套接得到额外的功能。 |
| FilterOutputStream | 实现装饰器功能的抽象类。为其它OutputStream对象增加额外的功能。 | 见下表 |
| 见下表 |

       装饰输出字节流：

|  |  |  |
| --- | --- | --- |
| 类 | 功能 | 如何构造 |
| 怎样使用 |
| DataOutputStream | 通常和DataInputStream配合使用，使用它可以写入基本数据类新。 | 使用OutputStream构造 |
| 包含大量的写入基本数据类型的方法。 |
| PrintStream | 产生具有格式的输出信息。（一般地在java程序中DataOutputStream用于数据的存储，即J2EE中持久层完成的功能，PrintStream完成显示的功能，类似于J2EE中表现层的功能） | 使用OutputStream和一个可选的表示缓存是否在每次换行时是否flush的标记构造。还提供很多和文件相关的构造方法。 |
| 一般是一个终极（“final”）的包装器，很多时候我们都使用它！ |
| BufferedOutputStream | 使用它可以避免频繁地向IO写入数据，数据一般都写入一个缓存区，在调用flush方法后会清空缓存、一次完成数据的写入。 | 从一个OutputStream或者和一个代表缓存区大小的可选参数构造。 |
| 提供和其它OutputStream一致的接口，只是内部提供一个缓存的功能。 |

### 3.3字节流的输入与输出的对应

       在3.1节讲过输入与输出的对应，下图表示字节流部分的输入与输出的对应关系。



上图中蓝色的为主要的对应部分，红色的部分就是不对应部分。我习惯上称之为“不入流”部分。紫色的虚线部分代表这些流一般要搭配使用。从上面的图中可以看出Java IO中的字节流是极其对称的。

“存在及合理”我们看看这些字节流中不太对称的几个类吧！

1.         LineNumberInputStream主要完成从流中读取数据时，会得到相应的行号，至于什么时候分行、在哪里分行是由改类主动确定的，并不是在原始中有这样一个行号。在输出部分没有对应的部分，我们完全可以自己建立一个LineNumberOutputStream，在最初写入时会有一个基准的行号，以后每次遇到换行时会在下一行添加一个行号，看起来也是可以的。好像更不入流了。

2.         PushbackInputStream的功能是查看最后一个字节，不满意就放入缓冲区。主要用在编译器的语法、词法分析部分。输出部分的BufferedOutputStream几乎实现相近的功能。

3.         StringBufferInputStream已经被Deprecated，本身就不应该出现在InputStream部分，主要因为String应该属于字符流的范围。已经被废弃了，当然输出部分也没有必要需要它了！还允许它存在只是为了保持版本的向下兼容而已。

4.         SequenceInputStream可以认为是一个工具类，将两个或者多个输入流当成一个输入流依次读取。完全可以从IO包中去除，还完全不影响IO包的结构，却让其更“纯洁”――纯洁的Decorator模式。

5.         PrintStream也可以认为是一个辅助工具。主要可以向其他输出流，或者FileInputStream写入数据，本身内部实现还是带缓冲的。本质上是对其它流的综合运用的一个工具而已。一样可以踢出IO包！System.out和System.out就是PrintStream的实例！

蓝色的部分是IO字节流的主要组成部分，存在极强的对称关系。关于搭配使用的三对类补充一下：ObjectInputStream/ObjectOutputStream和DataInputStream/DataOutputStream主要是要求写对象/数据和读对象/数据的次序要保持一致，否则轻则不能得到正确的数据，重则抛出异常(一般会如此)；PipedInputStream/PipedOutputStream在创建时一般就一起创建，调用它们的读写方法时会检查对方是否存在，或者关闭！道理极其简单――对方都不在了，怎么交互啊！

3.4 字节流与字符流

       从上面我们可以看出IO中的字节流是极其复杂的，存在大量的类，到目前为止还没有真正使用它们，使用它们应该也是极其复杂的吧！JDK1.1后Sun对IO库进行了重大的改进。看到Reader和Writer类时，大多数人的第一个感觉（不要太相信感觉哦！感觉也许会欺骗你的！）就是它们是用来替换原来的InputStream和OutputStream类。有新的类，干吗还使用旧的呢！？但实情并非如此。尽管Sun不建议使用原始的流库中的某些功能，但原来的流依然得到了保留，不仅为了保持向后兼容，主要原因是新库不是旧库的替代，而是对旧库的增强。从以下两点可以明显地看出：

(1) 在老式的类层次结构里加入了新的类，这表明 Sun公司没有放弃老式流库的意图。

(2) 在许多情况下，新库中类的使用需要联合老结构中的类。为达到这个目的，需要使用一些“桥”类，如：InputStreamReader将一个InputStream转换成Reader；OutputStreamWriter将一个OutputStream转换成Writer。

那么Sun为什么在Java 1.1里添加了Reader和Writer层次，最重要的原因便是国际化（Internationalization――i18n）的需求。老式IO流层次结构只支持8位字节流，不能很好地控制16位的Unicode字符。Java本身支持Unicode，Sun又一致吹嘘其支持Unicode，因此有必要实现一个支持Unicode的流的层次结构，所以出现了Reader和Writer层次，以提供对所有IO操作中的Unicode的支持。除此之外，新库也对速度进行了优化，可比旧库更快地运行。

8位的字节流和16位的字符流的对应关系，可以从ByteInputStream/ByteOutputStream与CharArrayInputStream/CharArrayOutputStream的对应关系中看出端倪。（还没看出来啊！赶紧去看看Java的基本数据类型）。

因此在Java的IO体系中存在字节流和字符流的对应关系。下面就看看字符流吧！

3.5 IO中的输入字符流

       下面是IO中输入字符流的继承图。

* + Reader
    - BufferedReader
      * LineNumberReader
    - CharArrayReader
    - FilterReader
      * PushbackReader
    - InputStreamReader
      * FileReader
    - PipedReader
    - StringReader

在上面的关系图中可以看出：1.Reader是所有的输入字符流的父类，它是一个抽象类。2.CharReader、StringReader是两种基本的介质流，它们分别将Char数组、String中读取数据。PipedReader是从与其它线程共用的管道中读取数据。3. BufferedReader很明显就是一个装饰器，它和其子类负责装饰其它Reader对象。4.FilterReader是所有自定义具体装饰流的父类，其子类PushbackReader对Reader对象进行装饰，会增加一个行号。5.InputStreamReader是一个连接字节流和字符流的桥梁，它将字节流转变为字符流。FileReader可以说是一个达到此功能、常用的工具类，在其源代码中明显使用了将FileInputStream转变为Reader的方法。我们可以从这个类中得到一定的技巧。

       Reader中各个类的用途和使用方法基本和InputStream中的类使用一致。后面会有Reader与InputStream的对应关系。

3.6 IO中的输出字符流

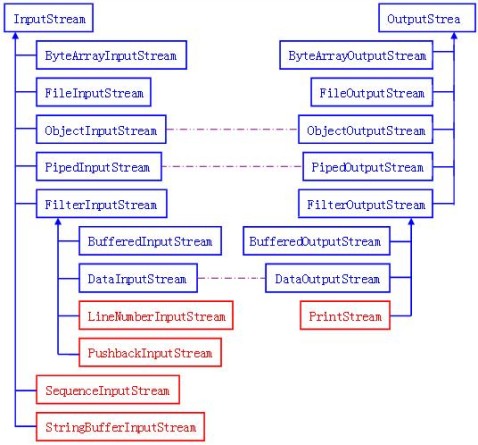
下面是IO中输出字符流的继承图。

* + Writer
    - BufferedWriter
    - CharArrayWriter
    - FilterWriter
    - OutputStreamWriter
      * FileWriter
    - PipedWriter
    - PrintWriter
    - StringWriter

在上面的关系图中可以看出：1.Writer是所有的输出字符流的父类，它是一个抽象类。2. CharArrayWriter、StringWriter是两种基本的介质流，它们分别向Char数组、String中写入数据。PipedWriter是向与其它线程共用的管道中写入数据， 3. BufferedWriter是一个装饰器为Writer提供缓冲功能。4.PrintWriter和PrintStream极其类似，功能和使用也非常相似。5.OutputStreamWriter是OutputStream到Writer转换的桥梁，它的子类FileWriter其实就是一个实现此功能的具体类（具体可以研究一下Source Code）。功能和使用和OutputStream极其类似，后面会有它们的对应图。

3.7字符流的输入与输出的对应

下图为字符流的输入与输出的对应关系图：



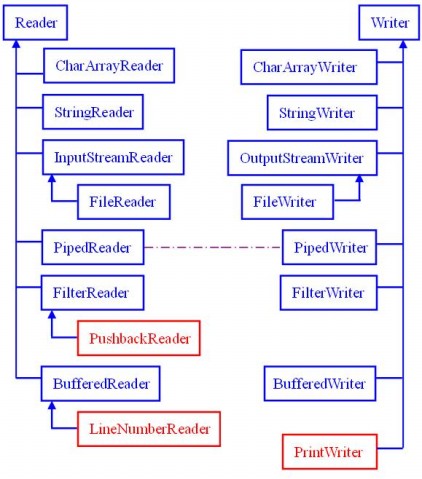
对应关系和字节流的输入输出基本一致，不必多说了吧！在下面的源代码阅读部分会仔细研究一些！

3.8字节流和字符流的对应

       Java的IO中存在输入、输出的对应和字节流和字符流的对应，下面就看看字节流和字符流的对应吧！

3.8.1输入的对应

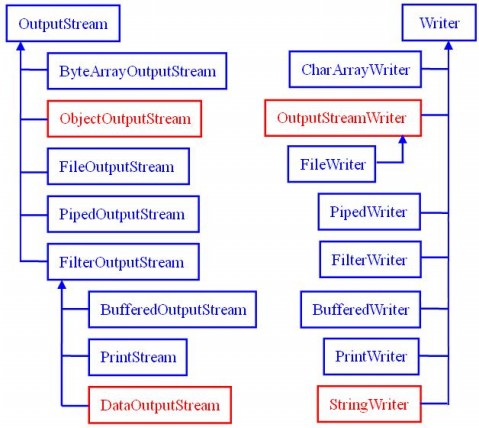
       下图是IO中字节输入流与字符输入流的对应图：



       蓝色的表示对应的部分，红色的表示不对应的部分。至于为什么不对应还是你自己多看看源代码、多考虑考虑吧！还要强调一点就是即使对应，它们的继承关系也是不太对应的。

3.8.2输出的对应

下图是IO中字节输出流与字符输出流的对应图：



       不多说了！等讲述了Adapter和Decorator模式会基本明白IO架构的！通过几个实例一般就可以使用了！

## 从InputStream到ByteArrayInputStream

    本篇主要分析：1.如何将byte数组适配至ByteArrayInputStream，对应与IO部分的适配器模式；2.BufferedInputStream的工作原理，对应于IO的装饰器模式，会首先研究InputStream和FilterInputStream的源代码，同时会将要谈谈软件设计中的缓存相关的知识。后面专门一章分析PipedInputStream和PipedOutStream，简单谈谈管道相关的知识，以及软件架构的想法。

1 InputStream

InputStream 是输入字节流部分，装饰器模式的顶层类。主要规定了输入字节流的公共方法。

package java.io;

public abstract class InputStream implements Closeable {

    private static final int SKIP\_BUFFER\_SIZE = 2048;  //用于skip方法，和skipBuffer相关

    private static byte[] skipBuffer;    // skipBuffer is initialized in skip(long), if needed.

  public abstract int read() throws IOException;

//从输入流中读取下一个字节，

//正常返回0-255，到达文件的末尾返回-1

//在流中还有数据，但是没有读到时该方法会阻塞（block）

//Java IO和New IO的区别就是阻塞流和非阻塞流

 //抽象方法哦！不同的子类不同的实现哦！

//将流中的数据读入放在byte数组的第off个位置先后的len个位置中

//放回值为放入字节的个数。

    public int read(byte b[], int off, int len) throws IOException {           //

         if (b == null) {

             throw new NullPointerException();

         } else if (off < 0 || len < 0 || len > b.length - off) {

             throw new IndexOutOfBoundsException();

         } else if (len == 0) {

             return 0;

         }        //检查输入是否正常。一般情况下，检查输入是方法设计的第一步

         int c = read();                                //读取下一个字节

         if (c == -1) {    return -1;   }           //到达文件的末端返回-1

         b[off] = (byte)c;                            //放回的字节downcast

         int i = 1;                                        //已经读取了一个字节

         try {

             for (; i < len ; i++) {                 //最多读取len个字节，所以要循环len次

                   c = read();                   //每次循环从流中读取一个字节

                                                        //由于read方法阻塞，

//所以read(byte[],int,int)也会阻塞

                   if (c == -1) {

  break;

}       //到达末尾，理所当然放回-1

                   b[off + i] = (byte)c;           //读到就放入byte数组中

             }

         } catch (IOException ee) {     }

         return i;

         //上面这个部分其实还有一点比较重要，int i = 1;在循环的外围，或许你经常见到，

         //或许你只会在循环是才声明，为什么呢？

         //声明在外面，增大了变量的生存周期（在循环外面），所以后面可以return返回

         //极其一般的想法。在类成员变量生命周期中使用同样的理念。

         //在软件设计中，类和类的关系中也是一样的。

    }        //这个方法在利用抽象方法read，某种意义上简单的Templete模式。

    public int read(byte b[]) throws IOException {

                   return read(b, 0, b.length);

    }                           //利用上面的方法read(byte[] b)

    public long skip(long n) throws IOException {

         long remaining = n;                                  //方法内部使用的、表示要跳过的字节数目，

//使用它完成一系列字节读取的循环

         int nr;

         if (skipBuffer == null)

             skipBuffer = new byte[SKIP\_BUFFER\_SIZE];                   //初始化一个跳转的缓存

         byte[] localSkipBuffer = skipBuffer;                                      //本地化的跳转缓存

         if (n <= 0) {    return 0;      }                           //检查输入参数，应该放在方法的开始

         while (remaining > 0) {                                      //一共要跳过n个，每次跳过部分，循环

             nr = read(localSkipBuffer, 0, (int) Math.min(SKIP\_BUFFER\_SIZE, remaining));

                                                        //利用上面的read(byte[],int,int)方法尽量读取n个字节

             if (nr < 0) {  break;    }                          //读到流的末端，则返回

             remaining -= nr;                                       //没有完全读到需要的，则继续循环

         }

         return n - remaining;//返回时要么全部读完，要么因为到达文件末端，读取了部分

    }

    public int available() throws IOException {                  //查询流中还有多少可以读取的字节

                   return 0;

    }

         //该方法不会block。在java中抽象类方法的实现一般有以下几种方式：

//1.抛出异常（java.util）；2.“弱”实现。象上面这种。子类在必要的时候覆盖它。

//3.“空”实现。下面有例子。

    public void close() throws IOException {}

         //关闭当前流、同时释放与此流相关的资源

    public synchronized void mark(int readlimit) {}

         //在当前位置对流进行标记，必要的时候可以使用reset方法返回。

         //markSupport可以查询当前流是否支持mark

    public synchronized void reset() throws IOException {

                   throw new IOException("mark/reset not supported");

    }

         //对mark过的流进行复位。只有当流支持mark时才可以使用此方法。

         //看看mark、available和reset方法。体会为什么？！

    public boolean markSupported() {           //查询是否支持mark

                   return false;

    }                 //绝大部分不支持，因此提供默认实现，返回false。子类有需要可以覆盖。

}

2 FilterInputStream

       这是字节输入流部分装饰器模式的核心。是我们在装饰器模式中的Decorator对象，主要完成对其它流装饰的基本功能。下面是它的源代码:

package java.io;

//该类对被装饰的流进行基本的包裹。不增加额外的功能。

//客户在需要的时候可以覆盖相应的方法。具体覆盖可以在ByteInputStream中看到！

public class FilterInputStream extends InputStream {

    protected volatile InputStream in;                       //将要被装饰的字节输入流

    protected FilterInputStream(InputStream in) {   //通过构造方法传入此被装饰的流

                   this.in = in;

    }

         //装饰器的代码特征：被装饰的对象一般是装饰器的成员变量

         //上面几行可以看出。

         //下面这些方法，完成最小的装饰――0装饰，只是调用被装饰流的方法而已

    public int read() throws IOException {

                   return in.read();

    }

    public int read(byte b[]) throws IOException {

                   return read(b, 0, b.length);

    }

    public int read(byte b[], int off, int len) throws IOException {

                   return in.read(b, off, len);

    }

    public long skip(long n) throws IOException {

                   return in.skip(n);

    }

    public int available() throws IOException {

                   return in.available();

    }

    public void close() throws IOException {

                   in.close();

    }

    public synchronized void mark(int readlimit) {

                   in.mark(readlimit);

    }

    public synchronized void reset() throws IOException {

                   in.reset();

    }

    public boolean markSupported() {

                   return in.markSupported();

}

//以上的方法，都是通过调用被装饰对象in完成的。没有添加任何额外功能

//装饰器模式中的Decorator对象，不增加被装饰对象的功能。

//它是装饰器模式中的核心。更多关于装饰器模式的理论请阅读博客中的文章。

}

       以上分析了所有字节输入流的公共父类InputStream和装饰器类FilterInputStream类。他们是装饰器模式中两个重要的类。更多细节请阅读博客中装饰器模式的文章。下面将讲解一个具体的流ByteArrayInputStream，不过它是采用适配器设计模式。

3 ByteArray到ByteArrayInputStream的适配

// ByteArrayInputStream内部有一个byte类型的buffer。

//很典型的适配器模式的应用――将byte数组适配流的接口。

//下面是源代码分析：

package java.io;

public class ByteArrayInputStream extends InputStream {

    protected byte buf[];                //内部的buffer，一般通过构造器输入

protected int pos;                   //当前位置的cursor。从0至byte数组的长度。

//byte[pos]就是read方法读取的字节

    protected int mark = 0;           //mark的位置。

    protected int count;                          //流中字节的数目。不一定与byte[]的长度一致？？？

    public ByteArrayInputStream(byte buf[]) {//从一个byte[]创建一个ByteArrayInputStream

         this.buf = buf;                                                      //初始化流中的各个成员变量

        this.pos = 0;

         this.count = buf.length;                              //count就等于buf.length

    }

    public ByteArrayInputStream(byte buf[], int offset, int length) {                //构造器

         this.buf = buf;

        this.pos = offset;                                                                                      //与上面不同

         this.count = Math.min(offset + length, buf.length);

        this.mark = offset;                                                                                             //与上面不同

    }

    public synchronized int read() {                                           //从流中读取下一个字节

                   return (pos < count) ? (buf[pos++] & 0xff) : -1; //返回下一个位置的字节

                                                                                                                //流中没有数据则返回-1

    }

         //下面这个方法很有意思！从InputStream中可以看出其提供了该方法的实现。

         //为什么ByteArrayInputStream要覆盖此方法呢？

         //同样的我们在Java Collections Framework中可以看到：

//AbstractCollection利用iterator实现了Collecion接口的很多方法。但是，

//在ArrayList中却有很多被子类覆盖了。为什么如此呢？？

    public synchronized int read(byte b[], int off, int len) {

         if (b == null) {                                                               //首先检查输入参数的状态是否正确

             throw new NullPointerException();

         } else if (off < 0 || len < 0 || len > b.length - off) {

             throw new IndexOutOfBoundsException();

         }

         if (pos >= count) {             return -1;             }

         if (pos + len > count) {      len = count - pos;         }

         if (len <= 0) {           return 0;     }

         System.arraycopy(buf, pos, b, off, len);                     //java中提供数据复制的方法

         pos += len;

         return len;

    }

         //出于速度的原因！他们都用到System.arraycopy方法。想想为什么？

         //某些时候，父类不能完全实现子类的功能，父类的实现一般比较通用。

//当子类有更有效的方法时，我们会覆盖这些方法。这样可是不太OO的哦！

         //下面这个方法，在InputStream中也已经实现了。

//但是当时是通过将字节读入一个buffer中实现的，好像效率低了一点。

//看看下面这段代码，是否极其简单呢？！

    public synchronized long skip(long n) {

         if (pos + n > count) {    n = count - pos;       }        //当前位置，可以跳跃的字节数目

         if (n < 0) {       return 0;     }                                    //小于0，则不可以跳跃

         pos += n;                                                                              //跳跃后，当前位置变化

         return n;

    }                                    //比InputStream中的方法简单、高效吧！

    public synchronized int available() {

                   return count - pos;

    }

         //查询流中还有多少字节没有读取。

//在我们的ByteArrayInputStream中就是当前位置以后字节的数目。

    public boolean markSupported() {

                   return true;

    }        //ByteArrayInputStream支持mark所以返回true

    public void mark(int readAheadLimit) {

                   mark = pos;

    }

//在流中当前位置mark。

//在我们的ByteArrayInputStream中就是将当前位置赋给mark变量。

//读取流中的字节就是读取字节数组中当前位置向后的的字节。

    public synchronized void reset() {

                   pos = mark;

    }

         //重置流。即回到mark的位置。

    public void close() throws IOException {   }

         //关闭ByteArrayInputStream不会产生任何动作。为什么？仔细考虑吧！！

}

上面我们分3小节讲了装饰器模式中的公共父类（对应于输入字节流的InputStream）、Decorator（对应于输入字节流的FilterInputStream）和基本被装饰对象（对应于输入字节流的媒体字节流）。下面我们就要讲述装饰器模式中的具体的包装器（对应于输入字节流的包装器流）。

4 BufferedInputStream

4.1原理及其在软件硬件中的应用

       1.read――read(byte[] ,int , int)

       2.BufferedInputStream

       3.《由一个简单的程序谈起》

       4. Cache

       5.Pool

       6.Spling Printer

       （最近比较忙，不讲了！）

4.2 BufferedInputStream源代码分析

package java.io;

import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

//该类主要完成对被包装流，加上一个缓存的功能

public class BufferedInputStream extends FilterInputStream {

    private static int defaultBufferSize = 8192;                                      //默认缓存的大小

    protected volatile byte buf[];                                                            //内部的缓存

    protected int count;                                                                                            //buffer的大小

    protected int pos;                                                                               //buffer中cursor的位置

    protected int markpos = -1;                                                                     //mark的位置

    protected int marklimit;                                                                            //mark的范围

//原子性更新。和一致性编程相关

    private static final

        AtomicReferenceFieldUpdater<BufferedInputStream, byte[]> bufUpdater =

        AtomicReferenceFieldUpdater.newUpdater (BufferedInputStream.class,  byte[].class, "buf");

    private InputStream getInIfOpen() throws IOException {  //检查输入流是否关闭，同时返回被包装流

        InputStream input = in;

         if (input == null)    throw new IOException("Stream closed");

        return input;

    }

    private byte[] getBufIfOpen() throws IOException {                       //检查buffer的状态，同时返回缓存

        byte[] buffer = buf;

         if (buffer == null)   throw new IOException("Stream closed");            //不太可能发生的状态

        return buffer;

    }

    public BufferedInputStream(InputStream in) {                               //构造器

                   this(in, defaultBufferSize);                                                              //指定默认长度的buffer

    }

    public BufferedInputStream(InputStream in, int size) {                           //构造器

                   super(in);

        if (size <= 0) {                                                                                         //检查输入参数

            throw new IllegalArgumentException("Buffer size <= 0");

        }

                   buf = new byte[size];                                                                     //创建指定长度的buffer

    }

         //从流中读取数据，填充如缓存中。

    private void fill() throws IOException {

        byte[] buffer = getBufIfOpen();                            //得到buffer

         if (markpos < 0)

             pos = 0;                                                             //mark位置小于0，此时pos为0

         else if (pos >= buffer.length)                               //pos大于buffer的长度

             if (markpos > 0) {

                   int sz = pos - markpos;                            //

                   System.arraycopy(buffer, markpos, buffer, 0, sz);

                   pos = sz;

                   markpos = 0;

             } else if (buffer.length >= marklimit) {                 //buffer的长度大于marklimit时，mark失效

                   markpos = -1;                                                   //

                   pos = 0;                                                             //丢弃buffer中的内容

             } else {                                                                         //buffer的长度小于marklimit时对buffer扩容

                   int nsz = pos \* 2;

                   if (nsz > marklimit)           nsz = marklimit;//扩容为原来的2倍，太大则为marklimit大小

                   byte nbuf[] = new byte[nsz];

                   System.arraycopy(buffer, 0, nbuf, 0, pos);        //将buffer中的字节拷贝如扩容后的buf中

                if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {

                                                                                                                         //在buffer在被操作时，不能取代此buffer

                    throw new IOException("Stream closed");

                }

                buffer = nbuf;                                                               //将新buf赋值给buffer

             }

        count = pos;

         int n = getInIfOpen().read(buffer, pos, buffer.length - pos);

        if (n > 0)     count = n + pos;

    }

    public synchronized int read() throws IOException { //读取下一个字节

         if (pos >= count) {                                                                 //到达buffer的末端

             fill();                                                                    //就从流中读取数据，填充buffer

             if (pos >= count)  return -1;                                //读过一次，没有数据则返回-1

         }

         return getBufIfOpen()[pos++] & 0xff;                           //返回buffer中下一个位置的字节

    }

    private int read1(byte[] b, int off, int len) throws IOException {                 //将数据从流中读入buffer中

         int avail = count - pos;                                                                             //buffer中还剩的可读字符

         if (avail <= 0) {                                                                                        //buffer中没有可以读取的数据时

             if (len >= getBufIfOpen().length && markpos < 0) {             //将输入流中的字节读入b中

                   return getInIfOpen().read(b, off, len);

             }

             fill();                                                                                                //填充

             avail = count - pos;

             if (avail <= 0) return -1;

         }

         int cnt = (avail < len) ? avail : len;                                                  //从流中读取后，检查可以读取的数目

         System.arraycopy(getBufIfOpen(), pos, b, off, cnt);            //将当前buffer中的字节放入b的末端

         pos += cnt;

         return cnt;

    }

    public synchronized int read(byte b[], int off, int len)throws IOException {

        getBufIfOpen();                                                                             // 检查buffer是否open

        if ((off | len | (off + len) | (b.length - (off + len))) < 0) {            //检查输入参数是否正确

             throw new IndexOutOfBoundsException();

         } else if (len == 0) {

            return 0;

        }

         int n = 0;

        for (;;) {

            int nread = read1(b, off + n, len - n);

            if (nread <= 0)     return (n == 0) ? nread : n;

            n += nread;

            if (n >= len)     return n;

            // if not closed but no bytes available, return

            InputStream input = in;

            if (input != null && input.available() <= 0)     return n;

        }

    }

    public synchronized long skip(long n) throws IOException {

        getBufIfOpen();                                        // 检查buffer是否关闭

         if (n <= 0) {    return 0;      }                 //检查输入参数是否正确

         long avail = count - pos;                    //buffered中可以读取字节的数目

        if (avail <= 0) {                                          //可以读取的小于0，则从流中读取

            if (markpos <0)  return getInIfOpen().skip(n); //mark小于0，则mark在流中

            fill();                                  // 从流中读取数据，填充缓冲区。

            avail = count - pos;                                   //可以读的取字节为buffer的容量减当前位置

            if (avail <= 0)     return 0;

        }

        long skipped = (avail < n) ? avail : n;

        pos += skipped;                                       //当前位置改变

        return skipped;

    }

    public synchronized int available() throws IOException {

                   return getInIfOpen().available() + (count - pos);

    }

         //该方法不会block！返回流中可以读取的字节的数目。

         //该方法的返回值为缓存中的可读字节数目加流中可读字节数目的和

    public synchronized void mark(int readlimit) {  //当前位置处为mark位置

         marklimit = readlimit;

         markpos = pos;

    }

    public synchronized void reset() throws IOException {

        getBufIfOpen(); // 缓冲去关闭了，肯定就抛出异常！程序设计中经常的手段

                   if (markpos < 0)     throw new IOException("Resetting to invalid mark");

                   pos = markpos;

    }

    public boolean markSupported() {           //该流和ByteArrayInputStream一样都支持mark

                   return true;

    }

         //关闭当前流同时释放相应的系统资源。

    public void close() throws IOException {

        byte[] buffer;

        while ( (buffer = buf) != null) {

            if (bufUpdater.compareAndSet(this, buffer, null)) {

                InputStream input = in;

                in = null;

                if (input != null)    input.close();

                return;

            }

            // Else retry in case a new buf was CASed in fill()

        }

    }

}

## 从PipedInputStream/PipedOutputStream谈起

       本篇主要从分析PipeInputStrem和PipedOutputStream谈起。谈及软件设计的变化，以及如何将软件拆分、组合，适配……

1 源代码分析

       下面将详细分析PipedInputStream和PipedOutputStream的源代码。

1.1 PipedInputStream

package java.io;

//PipedInputStream必须和PipedOutputStream联合使用。即必须连接输入部分。

//其原理为：PipedInputStream内部有一个Buffer，

//PipedInputStream可以使用InputStream的方法读取其Buffer中的字节。

//PipedInputStream中Buffer中的字节是PipedOutputStream调用PipedInputStream的方法放入的。

public class PipedInputStream extends InputStream {

    boolean closedByWriter = false;                                                             //标识有读取方或写入方关闭

    volatile boolean closedByReader = false;

    boolean connected = false;                                                                     //是否建立连接

    Thread readSide;                                                                                             //标识哪个线程

    Thread writeSide;

    protected static final int PIPE\_SIZE = 1024;                         //缓冲区的默认大小

    protected byte buffer[] = new byte[PIPE\_SIZE];                  //缓冲区

    protected int in = -1;               //下一个写入字节的位置。0代表空，in==out代表满

    protected int out = 0;               //下一个读取字节的位置

    public PipedInputStream(PipedOutputStream src) throws IOException {                //给定源的输入流

                   connect(src);

    }

    public PipedInputStream() {    }                                                //默认构造器，下部一定要connect源

    public void connect(PipedOutputStream src) throws IOException {               //连接输入源

                   src.connect(this);                                                                           //调用源的connect方法连接当前对象

    }

    protected synchronized void receive(int b) throws IOException {                   //只被PipedOuputStream调用

        checkStateForReceive();                                                                                 //检查状态，写入

        writeSide = Thread.currentThread();                                                      //永远是PipedOuputStream

        if (in == out)     awaitSpace();                                                           //输入和输出相等，等待空间

         if (in < 0) {

             in = 0;

             out = 0;

         }

         buffer[in++] = (byte)(b & 0xFF);                                                             //放入buffer相应的位置

         if (in >= buffer.length) {      in = 0;         }                                             //in为0表示buffer已空

    }

    synchronized void receive(byte b[], int off, int len)  throws IOException {

        checkStateForReceive();

        writeSide = Thread.currentThread();                                   //从PipedOutputStream可以看出

        int bytesToTransfer = len;

        while (bytesToTransfer > 0) {

            if (in == out)    awaitSpace();                                 //满了，会通知读取的；空会通知写入

            int nextTransferAmount = 0;

            if (out < in) {

                nextTransferAmount = buffer.length - in;

            } else if (in < out) {

                if (in == -1) {

                    in = out = 0;

                    nextTransferAmount = buffer.length - in;

                } else {

                    nextTransferAmount = out - in;

                }

            }

            if (nextTransferAmount > bytesToTransfer)     nextTransferAmount = bytesToTransfer;

            assert(nextTransferAmount > 0);

            System.arraycopy(b, off, buffer, in, nextTransferAmount);

            bytesToTransfer -= nextTransferAmount;

            off += nextTransferAmount;

            in += nextTransferAmount;

            if (in >= buffer.length) {     in = 0;      }

        }

    }

    private void checkStateForReceive() throws IOException {                           //检查当前状态，等待输入

        if (!connected) {

            throw new IOException("Pipe not connected");

        } else if (closedByWriter || closedByReader) {

             throw new IOException("Pipe closed");

         } else if (readSide != null && !readSide.isAlive()) {

            throw new IOException("Read end dead");

        }

    }

    private void awaitSpace() throws IOException {                                              //Buffer已满，等待一段时间

         while (in == out) {                                                                                             //in==out表示满了，没有空间

             checkStateForReceive();                                                                       //检查接受端的状态

             notifyAll();                                                                                  //通知读取端

             try {

                 wait(1000);

             } catch (InterruptedException ex) {

                   throw new java.io.InterruptedIOException();

             }

         }

    }

    synchronized void receivedLast() {                  //通知所有等待的线程（）已经接受到最后的字节

         closedByWriter = true;                             //

         notifyAll();

    }

    public synchronized int read()  throws IOException {

        if (!connected) {                                                                              //检查一些内部状态

            throw new IOException("Pipe not connected");

        } else if (closedByReader) {

             throw new IOException("Pipe closed");

         } else if (writeSide != null && !writeSide.isAlive()&& !closedByWriter && (in < 0)) {

            throw new IOException("Write end dead");

         }

        readSide = Thread.currentThread();                                            //当前线程读取

         int trials = 2;                                                                                             //重复两次？？？？

         while (in < 0) {

             if (closedByWriter) {              return -1;        }                 //输入断关闭返回-1

             if ((writeSide != null) && (!writeSide.isAlive()) && (--trials < 0)) {          //状态错误

                   throw new IOException("Pipe broken");

             }

             notifyAll();                                                             // 空了，通知写入端可以写入

             try {

                 wait(1000);

             } catch (InterruptedException ex) {

                   throw new java.io.InterruptedIOException();

             }

        }

         int ret = buffer[out++] & 0xFF;                                                        //

         if (out >= buffer.length) {             out = 0;                }

         if (in == out) {           in = -1;                 }                             //没有任何字节

         return ret;

    }

    public synchronized int read(byte b[], int off, int len)  throws IOException {

     if (b == null) {                                                                                 //检查输入参数的正确性

         throw new NullPointerException();

     } else if (off < 0 || len < 0 || len > b.length - off) {

         throw new IndexOutOfBoundsException();

     } else if (len == 0) {

         return 0;

     }

     int c = read();                                                                                 //读取下一个

     if (c < 0) {    return -1;       }                                             //已经到达末尾了，返回-1

     b[off] = (byte) c;                                                                    //放入外部buffer中

     int rlen = 1;                                                                            //return-len

     while ((in >= 0) && (--len > 0)) {                                          //下一个in存在，且没有到达len

         b[off + rlen] = buffer[out++];                                         //依次放入外部buffer

         rlen++;

         if (out >= buffer.length) {         out = 0;           }        //读到buffer的末尾，返回头部

         if (in == out) {     in = -1;      }               //读、写位置一致时，表示没有数据

     }

     return rlen;                                                                            //返回填充的长度

    }

    public synchronized int available() throws IOException {             //返回还有多少字节可以读取

         if(in < 0)

             return 0;                                                                                         //到达末端，没有字节

         else if(in == out)

             return buffer.length;                                                               //写入的和读出的一致，表示满

         else if (in > out)

             return in - out;                                                                                 //写入的大于读出

         else

             return in + buffer.length - out;                                                //写入的小于读出的

    }

    public void close()  throws IOException {                //关闭当前流，同时释放与其相关的资源

         closedByReader = true;                                             //表示由输入流关闭

        synchronized (this) {     in = -1;    }        //同步化当前对象，in为-1

    }

}

1.2 PipedOutputStream

// PipedOutputStream一般必须和一个PipedInputStream连接。共同构成一个pipe。

//它们的职能是：

package java.io;

import java.io.\*;

public class PipedOutputStream extends OutputStream {

    private PipedInputStream sink;                //包含一个PipedInputStream

    public PipedOutputStream(PipedInputStream snk)throws IOException {       //带有目的地的构造器

                   connect(snk);

    }

    public PipedOutputStream() {  }                      //默认构造器，必须使用下面的connect方法连接

    public synchronized void connect(PipedInputStream snk) throws IOException {

        if (snk == null) {                                                                    //检查输入参数的正确性

            throw new NullPointerException();

        } else if (sink != null || snk.connected) {

             throw new IOException("Already connected");

         }

         sink = snk;                                                                           //一系列初始化工作

         snk.in = -1;

         snk.out = 0;

        snk.connected = true;

    }

    public void write(int b) throws IOException {                        //向流中写入数据

        if (sink == null) {    throw new IOException("Pipe not connected");      }

         sink.receive(b);            //本质上是，调用PipedInputStream的receive方法接受此字节

    }

    public void write(byte b[], int off, int len) throws IOException {

        if (sink == null) {                                                                   //首先检查输入参数的正确性

            throw new IOException("Pipe not connected");

        } else if (b == null) {

             throw new NullPointerException();

         } else if ((off < 0) || (off > b.length) || (len < 0) || ((off + len) > b.length) || ((off + len) < 0)) {

             throw new IndexOutOfBoundsException();

         } else if (len == 0) {

             return;

         }

         sink.receive(b, off, len);                                                                 //调用PipedInputStream的receive方法接受

    }

    public synchronized void flush() throws IOException {                 //flush输出流

         if (sink != null) {

            synchronized (sink) {     sink.notifyAll();     } //本质是通知输入流，可以读取

         }

    }

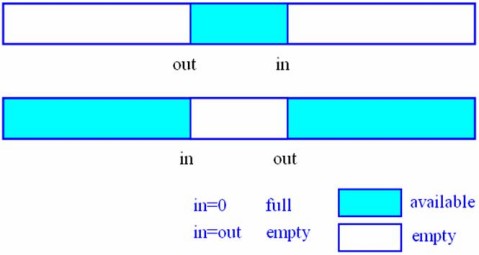
    public void close()  throws IOException {                         //关闭流同时释放相关资源

         if (sink != null) {    sink.receivedLast();         }

    }

}

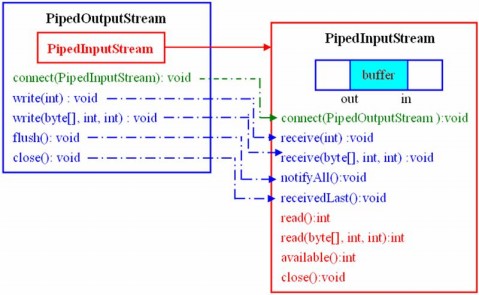
2 Buffer的状态



       上图是PipedInputStream中缓存的状态图。在程序中我们利用了byte数组，循环地向其中写入数据，写入有一个cursor（in），读出也有一个cursor（out）。上图表示in和out不同位置时，buffer中的各个位置的状态。蓝色的代表可以读取的字节。白色的表示此位置没有字节，或者此位置已经被PipedInputStream读取了。

3 交互简图

       下图是从源代码部分转换过来的关于PipedInputStream和PipedOutputStream的交互图。



       从图中可以看出：

1.         整个PipedInputStream是这对管道的核心。管道本身是一个byte的数组。

2.         PipedOutputStream对象通过Delegate方法复用PipedInputStream，同时屏蔽了其中的读取的方法，我们仅仅可以构造PipedOutputStream对象。（从这一点可以看出Delegate复用比继承复用的优越性了！）从设计模式的角度更象Adapter――PipedInputStream本身提供读取和写入的功能，将写入的功能适配到OutputStream，就成为一个PipedOutputStream。这样就形成一个类，适配后形成两种功能的类。

3.         调用PipedOutputStream的连接方法实际就是调用PipedInputStream的连接方法。

4.         调用PipedOutputStream的写相关的方法实际就是调用PipedInputStream的对应方法。

以上也是一种适配，将管道的概念适配到流的概念，同时将两者的职能分开。

4 将Chanel放入PipedOutputStream

       上面的例子中，Chanel放在PipedInputStream中，我们仔细思考后可以顺理成章地将其Chanel放入PipedOutputStream中。请注意synchronized方法是得到哪个字节流的锁！！

5 Chanel移出的一个例子

       在上面两个例子中Buffer要么在写入对象的内部，要么在读取对象的内部。主要通过适配该对象的方法，达到自己的需求而已。下面是一个一般的例子――将Chanel移出，Chanel提供了写入与读取的功能。这也完全合乎OO的“Single Responsibility Protocol――SRP”。输入部分使用Delegate复用此Chanel，将其适配至InputStream和OutputStream。下面是简单的Source code。

//PipedChanel.java

       import java.io.IOException ;

public class PipedChanel {

    protected static final int PIPE\_SIZE = 1024;

    protected byte buffer[] = new byte[PIPE\_SIZE];

    protected int in = -1;

protected int out = 0;

    public PipedChanel(){  }

    public PipedChanel(int size){

           buffer = new byte[size]  ;

    }

    public synchronized int read() throws IOException {    }

    public synchronized int read(byte b[], int off, int len)  throws IOException {    }

    public synchronized int available() throws IOException {}

    public synchronized void close()  throws IOException {}

public synchronized void write(int b)  throws IOException {}

public synchronized void write(byte b[]) throws IOException {}

    public synchronized void write(byte b[], int off, int len) throws IOException {}

public synchronized void flush() throws IOException {}

    public void waitWhileFull(){    }            //当Chanel已经满了，写线程等待

public void waitWhileEmpty{    }        //当Chanel为空，读取线程等待

//以上是两个操作Chanel时的状态相关的方法。

//是一致性编程部分，典型的设计模式。

//这两个方法，包含在对应读或写方法的最前面。

}

       // PipedChanelInputStream.java

import java.io.\*;

public class PipedChanelInputStream extends InputStream {

       private PipedChanel chanel ;

       public PipedChanelInputStream(PipedChanel chanel){

              this.chanel = chanel ;

       }

       public int read() throws IOException {

              return chanel.read();

       }

    public  int read(byte b[], int off, int len)  throws IOException {

           return chanel.read(b,off,len);

    }

    public  int available() throws IOException {

           return chanel.available();

    }

    public  void close()  throws IOException {

           chanel.close();

    }

}

       // PipedChanelOutputStream.java

import java.io.\*;

public class PipedChanelOutputStream extends OutputStream {

       private PipedChanel chanel ;

       public PipedChanelOutputStream(PipedChanel chanel){

              this.chanel = chanel ;

       }

    public synchronized void write(int b)  throws IOException {

           chanel.write(b);

    }

    public synchronized void write(byte b[]) throws IOException {

           chanel.write(b);

    }

    public synchronized void write(byte b[], int off, int len) throws IOException {

           chanel.write(b,off,len);

    }

    public synchronized void flush() throws IOException {

           chanel.flush();

    }

    public synchronized void close()  throws IOException {

           chanel.close();

    }

}

       很简单的例子。我们可以体会适配器模式，可以体会软件设计的灵活性……

       上面的关于PipedInputStream和PipedOutputStream的例子，本质上是对一个Chanel的几个不同的适配。Chanel作为一种编程模式，在软件设计中有极其广泛的应用。下面一节是JMS的简洁阐述！

       以上的例子其实是一个典型的使用适配器。

6 JMS的架构

       JMS为J2EE部分的面向消息中间件的API。JMS的Queue、Topic某种意义上就是我们上面Chanel移到网络的其它一段――服务器上的一个例子。同时该Chanel得到了很多强化。如：1.支持交易；2.支持持久化……

       在J2EE中JMS是一个比较重要的方向，大型的企业应用中都会使用。不过J2EE中给出了其API，背后的理念还是相当丰富的！